An Efficient Implementation of a Subgraph Isomorphism Algorithm for GPUs

Vincenzo Bonnici, Rosalba Giugno, and Nicola Bombieri, *Member, IEEE*

***Abstract*—The subgraph isomorphism problem is a computational task that applies to a wide range of today's applications, ranging from the understanding of biological networks to the analysis of social networks. Even though different implementations for CPUs have been proposed to improve the efficiency of such a graph search algorithm, they have shown to be bounded by the intrinsic sequential nature of the algorithm. More recently, graphics processing units (GPUs) have become widespread platforms that provide massive parallelism at low cost. Nevertheless, parallelizing any efficient and optimized sequential algorithm for subgraph isomorphism on many-core architectures is a very challenging task. This article presents *GRASS*, a parallel implementation of the subgraph isomorphism algorithm for GPUs. Different strategies are implemented in *GRASS* to deal with the space complexity of the graph searching algorithm, the potential workload imbalance, and the thread divergence involved by the non-homogeneity of actual graphs. The paper presents the results obtained on several graphs of different sizes and characteristics to understand the efficiency of the proposed approach.**

***Index Terms*—Subgraph isomorphism, Graph search, Parallel computing, GPU, CUDA.**

I. Introduction

Graphs are simple and powerful data structures to represent data objects and relations among them. They find a wide range of applications from Bioinformatics to Social Science. In Bioinformatics, biological networks are modelled through graphs where vertices are proteins and edges are the known or predicted protein interactions [1], [2]. In Pharmacology, 2D/3D representation of drugs are modelled through graphs where vertices are atoms and edges are the distances among them in the molecule [3], [4]. In Social Science, the Facebook network is represented by a graph that collects individuals or events on vertices and friendship relationships or individual preferences on edges [5].

Subgraph isomorphism (SubGI) is one of the most common algorithms applied to extrapolate and analyse information from graphs. Given a *query* graph Q and a *target* graph G, a SubGI algorithm searches occurrences of Q in G. Such a searching algorithm allows one to find, for example, *motifs* in biological networks [6], druggable parts of proteins [7], circuits in social and biological graphs [8], [9].

Even though several sequential and optimized solutions have been proposed in literature [10], [11], [12], [13], [14], [15] to deal with the complexity of the SubGI problem (which is known to be NP-complete [16]), the searching time still remains a big issue when dealing with realistically large graphs [17], [18], [19], [20]. In this perspective, the spreading of massively parallel architectures such as the general purpose graphic processing units (GPUs) may significantly reduce the run time of such a computational intensive algorithm by exploiting parallel execution. Several parallel implementations of other graph algorithms for GPUs, such as, breadth-first search (BFS) [21], [22], single-source shortest path (SSSP) [23], all-pairs shortest path (APSP) [24], have confirmed the potentiality of many-core architectures in speeding up the execution times. On the other hand, GPUs have also shown significant limitations when the algorithm implementations are inherently sequential, do not expose sufficient amount of fine-grained parallelism and memory access patterns with high locality, and do not allow the GPU cores to be fully utilized through *thread convergence* [25], [26], [27]. In this context, the serialized nature of the depth-first search (DFS), which is the core primitive of the SubGI algorithm in the sequential implementations, makes parallelization for many-core architectures a very challenging task. Thread divergence, workload imbalance, and poorly coalesced memory accesses are representative issues that arise when dealing with parallel search on graphs [28]. In addition, the memory footprint, which increases exponentially with the graph size, is a further hard constraint that may limit GPU implementations to work only on very small graphs [29]. These limitations highly influence current approaches to SubGI developed on GPU, which can deal with billion nodes target graphs but are limited to query graphs in the order of 10 nodes [30].

This paper presents *GRASS*, a parallel implementation of the SubGI algorithm for GPUs. *GRASS* overcomes the above limitations in two steps. First, it implements a pre-processing phase, in which heuristics are applied to sensibly reduce the search space. This leads *GRASS* to support even large graphs (i.e., of sizes around ten millions nodes and one hundred million edges) also when run on low-end GPU devices. Then, it combines DFS and BFS visiting strategies through two different kernels, to better exploit the massive thread parallelism and to optimize the workload balancing during the visit of non-homogeneous topologies of actual graphs. *GRASS* is available for download at *http://profs.scienze.univr.it/bombieri/GRASS*.

The main contributions of the paper are the following: (i) it presents a parallel implementation of the SubGI algorithm for GPU; (ii) it shows how a filtering-based strategy (e.g., [12]) can be adopted to deal with the space complexity of the problem. In particular, the paper shows which isomorphism conditions can be checked (preliminary and sequentially on the CPU) to filter the solution space and to allow adopting

The authors are affiliated with the Dipartimento di Informatica, Universita di Verona, Strata le Grazie, 15, 37134 Verona, Italy. E-mail: {vincenzo.bonnici, rosalba.giugno, nicola.bombieri}@univr.it

1: INPUT:$Q(V_Q, E_Q), G(V_G, E_G)$
2: $n = |V_Q|$
3: C = $\{(v_{i_1}, \ldots, v_{i_n}), v_{i_k} \in |V_G|, v_{i_k} \neq v_{i_j}, \forall 1 \leq k, j \leq n\}$ //Generate all lists of n distinct vertices of G
4: **for** s=1 **to** $|C|$ **do**
5: $M_s = ((u_1, v_{s_1}), \ldots, (u_n, v_{s_n}))$, $u_i \in |V_Q|$, $1 \leq i \leq n$ //Build mappings by associating to each vertex in Q a vertex in the list
6: **for all** $(u_i, u_j) \in |E_Q|$ **do**
7: $v_{s_i} = M_s(u_i)$
8: $v_{s_j} = M_s(u_j)$ // M_s is subgraph isomorphism if for all edges in Q, the mapped edges in G satisfy the isomorphim condition in Definition 1
9: **if** $(v_{s_i}, v_{s_j}) \notin E_G$ OR (v_{s_i}, v_{s_j}) does not satisfy the isomorphism conditions **then**
10: break
11: **end if**
12: **end for**
13: **return** M_s is a SubGI of Q in G
14: **end for**

Fig. 1. A SubGI algorithm

GPU devices, and which isomorphism condition is worth to be parallelized on the GPU; (iii) since SubGI relies on DFS, which is inherently sequential and not suitable for parallel execution on GPUs, the paper shows how to combine DFS and BFS to perform a *limited-depth BFS search*; (iv) it presents a performance analysis on several graphs of different sizes and characteristics, both biological and synthetic, to understand the correlation between performance and graph characteristics as well as advantages and limitations of the proposed approach.

II. Background and related work

Given a graph $G = (V, E)$, where V is the set of *vertices* and $E \subseteq (V \times V)$ is the set of *edges*, the properties of the modelled objects and their relationships are associated to vertices and edges, respectively, through *labels*. Given the set of labels A, the functions $lab_V : V \rightarrow A$ and $lab_E : E \rightarrow A$ assign labels to vertices and edges. If $(u, u') \in E$, u' is called a neighbor of u. Searching a query graph Q in a target graph G, $|Q| \leq |G|$, implies finding an injective function (i.e., subgraph isomorphism) $M : V_Q \rightarrow V_G$, which maps each vertex of Q into a unique vertex of G such that the following conditions hold:

Definition 1 (Isomorphism conditions). If (u, u') is an edge in Q, u has label $lab(u)$, u' has $lab(u')$, then the mapped pair of vertices $(M(u), M(u'))$ is an edge in G and has $lab_V(u) = lab_V(M(u))$, $lab_V(u') = lab_V(M(u'))$, and $lab_E(u, u') = lab_E(M(u), M(u'))$.

An algorithm implementing SubGI between Q and G searches all possible mappings between the vertices of the two graphs and checks whether any generated mapping satisfies the conditions (see Figure 1). When this happens, such a mapping is a SubGI of Q in G (which is called *occurrence* or *match*).

The solution space of all possible mappings is represented by the *search space tree*. The tree has a dummy root and each other node represents a mapping between a vertex u of the query Q and a vertex v of the target graph G (i.e., each node of the tree represents two vertices, the first of Q and the second of G)[1]. Figure 2 includes an example of search space tree, which will be discussed in detail in the following sections. Any path from the root to a leaf node may represent a match between Q and G. In particular, in a path if all the nodes between the root and the leaf satisfy the conditions, the path represents a match. If a node (not leaf) and its ancestors satisfy the conditions, the path from the root to such a node represents a *partial match*. The search space tree allows all possible mappings to be represented in a compact way. Two different match paths may share the same partial match path (e.g., the path from root to node $< 2|2 >$ is the common partial match for the two match instances in Figure 2).

The algorithm visits, in depth-first way, all the paths of the search space tree, starting from the root down to the leaves. For each path, at each node, if the conditions are not satisfied the algorithm prunes the underling branches and backtracks on the parent nodes.

The main issue to face when implementing the SubGI algorithm is the solution space (i.e., the size of the search space tree). In literature, several strategies have been proposed in the last decades to reduce the number of the tree paths [31], [14], [15], [32], [12], [13]. A large class of them run a pre-processing phase to build, for each vertex u of the query, a set of *matchable* vertices $\{v\}$ (called *domain*) of the target [13], [15], [33]. Vertices of the target must have the same or compatible labels of the query vertex. Moreover, topological conditions must hold such as the number of edges incident to u (called *degree*) must be less or equal to the number of vertices incident to v. For directed graphs this step verifies the above conditions for coming-out (out-degree) and going-in (in-degree) edges of the vertices. Other solutions force the solution space reduction by applying forehead rules to check whether the current match does not imply topological violations in the subsequent matches [13], [14], [15].

GPU-based methodologies have been developed for graph motif discovery. This problem highly relates with the SubGI, since motifs are extracted by searching for subgraph isomorphism. However, such a massive graph ming may better exploit parallel solutions as each run requires either the search of multiple patterns or the search of one single pattern in a dataset of targets. The two existing GPU approaches [29], [34] are based on the famous mining algorithm *gSpan* [35]. The first approach [29] has been tested on a dataset of very small target graphs (up to 40 vertices) but it showed a maximum speed-up of 80x. The second solution [34] scans for multiple patterns concurrently and can reach up to 15x speed-up for target graphs with around 20 vertices, while up to 3x speed-up for target graphs with 100k vertices.

In the context of 1-to-1 SubGI search, that is th class of algorithm our proposed approach belongs to, the authors in [36] show how one of the three main steps of the algorithm proposed in [10] (i.e., the join phase) can be efficiently parallelized on GPUs. They show that such a parallelization can lead to one order of magnitude of performance improvement w.r.t. *STWig* [10]. AS for the original implementation made

[1]In this work, we use the term *node* for the solution space tree and the term *vertex* for graphs.

for cluster-based memory cloud, the GPU approach uses a *divide and conquer* paradigm by splitting the query graph into sub-structures, called *wings*, that partially overlap each other. Wings are searched separately in parallel and the results are joined after the parallel search. Wings are 1-neighborhood substructures that do not require particular techniques to be matched on GPU, while the authors developed an innovative GPU joint algorithm based on hash tables. The new technique allows reaching speed-ups up to 16.7x on SubGI instances for target and query graphs having up to 128k and 10 vertices, respectively, when adopting standard GPU devices ($\simeq$ 10GB DRAM memory). A more refined approach, but still based on the *STWig* methodology, was proposed in [30]. The new GPU methodology, called *GpSM*, takes the edges as the basic units of the problem and adopts a pruning technique [37], that ignores low-connected vertices, to reduce the number of intermediate results. The approach dynamically explores the search space to improve the workload balancing. It showed a speed-up up to 5x compared to *TurboISO* on target graphs with 100k vertices and 24 vertices patterns. The authors showed that it can search on graphs with 2 billion vertices but with limited pattern sizes, up to 13 vertices. *GpSM* scales from 100k to 2 billions vertices (target graphs) with a factor of 14x.

Divide and conquer approaches have also been proposed for the specific problem of the graph isomorphism. In [38], explicit join operations are avoided and the algorithm completely relies on the graph traversal by adopting a warp-centric programming model to balance the workload among threads in warps. Such a GPU implementation reaches a speed-up up to 2.6x on instances having target and pattern graphs up to 4k and 24 vertices, respectively. A similar approach was proposed in [39] and tested on synthetic target graphs having up to 20k vertices. Nevertheless, no further details are given and the implementation is not available.

The approach proposed in this paper and implemented in *GRASS* overcomes the above limitations by splitting the verification of the isomorphism conditions in two steps. First, in a pre-processing phase, it builds a *reduced search space tree* by applying heuristics, as explained in Section III. Then, it combines DFS and BFS visiting strategies on the search tree to verify, in parallel, the isomorphism conditions along the paths of the reduced search tree, as explained in Section IV.

III. Reducing the search space

GRASS generates the reduced search space tree by identifying, in advance, the nodes of the tree that cannot satisfy the isomorphism conditions. A path containing at least one of those nodes cannot lead to isomorphism occurrences (i.e., matches) and, thus, it can be pruned. All remaining nodes are called *candidate* to isomorphism occurrences. The result of this first step is a reduced search space tree in which all nodes are *candidates* to represent partial matches (if not leaves) or matches (if leaves).

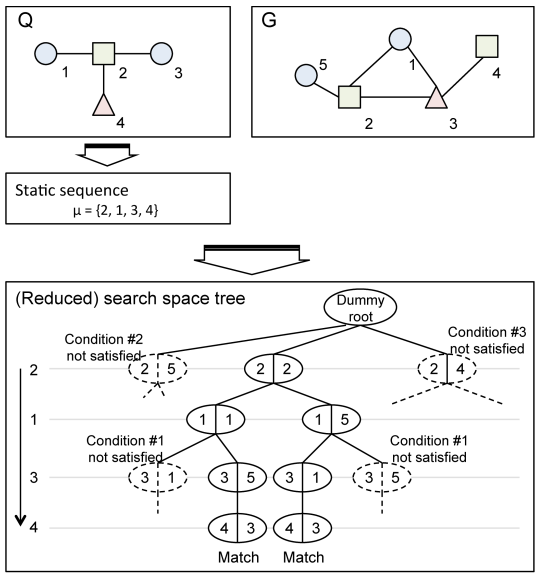

Fig. 2. Generation of the search space tree given the query Q and the graph G, with the proposed isomorphism conditions for search space reduction. The static sequence of vertices μ is the order with which query vertices are matched.

The candidate conditions are defined as follows[2]:

Definition 2 (Candidate conditions). Given u_i a vertex of Q and $M(u_i)$ its mapped vertex in G, (u_i,$M(u_i)$) is a candidate node of the search tree if:

1) Neither u_i nor $M(u_i)$ are already mapped in the current path.
2) *Label compatibility*. The mapped vertices are compatible, i.e., $lab(u_i) = lab(M(u_i))$.
3) *Degree compatibility*. The number of edges connected to $M(u_i)$ in V_G is greater than or equal to the number of edges connected to u_i in V_Q. That is $|\{(v', M(u_i)) \in E_G\}| \geq |\{(w, u_i) \in E_Q\}|and|\{(M(u_i), v') \in E_G\}| \geq |\{(u_i, w) \in E_Q\}|$.
4) *Neighbor degree compatibility*. Given u_i' the neighbor of u_i with the maximum degree and $M(u_i)$, $\exists v'$ s.t. v' is a neighbor of $M(u_i)$ that satisfies the degree compatibility with u_i'.

Figure 2 shows an example of search space tree generated starting from a query graph Q and a target graph G. The tree is reduced from the nodes (and the corresponding underlying branches) that do not satisfy the candidate conditions.

GRASS verifies the candidate conditions at the search tree generation time. Starting from the root, the search space tree is enriched with one node at a time. A new node is inserted if the node satisfies the conditions. If not, such a node and the corresponding underling branches are pruned. This allows the reduced space tree to be built incrementally and, as a consequence, the maximum peak of required resident memory to be drastically minimized.

[2]For the sake of clarity and without loss of generality, in the following definitions, we consider Q and G as connected graphs, we ignore edge labels, and we consider directed graphs (i.e., $(u, u') \in E$ does not imply $(u', u) \in E$). However, the proposed approach also applies to undirected connected graphs.

In particular, *GRASS* verifies the candidate condition i only if condition $i-1$ does not fail. The conditions are checked for all vertices of the query Q over all vertices of the target G, by generating a result matrix of $|V_Q| \times |V_G|$ size (*CandidateTree[][]*). The preprocessing phase reduces the search space from

$$\sum_{i=1}^{|V_Q|} \frac{|V_G|!}{(|V_G|-i)!} = \sum_{j=1}^{|V_Q|} \prod_{i=1}^{j} |V_G| - i + 1$$

to

$$\sum_{j=1}^{|V_Q|} \prod_{i=1}^{j} |C_i| - i + 1$$

where C_i is the set of candidates of u_i, and it holds that $|C_i| \ll |V_G|$.

In addition, *GRASS* applies the heuristic proposed in [12] to choose the order in which the query vertices are considered during the space tree generation. Such an order affects both the solution space size and the performance of the following step of isomorphism checking [15], [40], [12].

The order is defined by considering the degree of the query vertices and their connections with vertices already in the order. In this way, *GRASS* imposes a large number of subgraph matching conditions to be checked at the top nodes of the search tree.

Given a query graph $Q(V_Q, E_Q)$, let $n = |V_Q|$, the ordered sequence of query vertices μ=$(u_0, u_1, \ldots, u_n)$ of V_Q is generated by choosing at each step i, the vertex $u_i \in V_Q$ such that it maximizes the number of edges in the query that connects u_i with vertices in μ.

The chosen order is static for all the branches, in contrast to other sequential approaches in which the ordering may differ over different branches [15], [40]. A static ordering has been preferred since it requires less allocated memory than the dynamic one [12]. On the other hand, the advantages of the dynamic ordering, which are considerable in the sequential solutions, are compensated in the following parallel step.

The overall complexity of the reduced tree generation phase is $O(|V_Q| \times |V_G| \times \delta)$, where δ is the maximum degree of the graph (since condition 2.4 takes $O(\delta)$ to verify. It is negligible w.r.t. the overall searching time. *GRASS* runs this check sequentially on the CPU.

IV. The parallel search algorithm

In the second step, *GRASS* implements a visit of the reduced search space tree (hereafter called *tree*) and checks, for each node of the tree, whether the following constraint holds:

Definition 3 (Topology constraint). Given u_i a vertex of Q and $M(u_i)$ its mapped vertex in G, (u_i,$M(u_i)$) satisfies the topology constraint if $\forall u_i, u_j \in V_Q$ $(u_i, u_j) \in E_Q \Rightarrow (M(u_i), M(u_j)) \in E_G$. If edges are labeled the compatibility of the edge labels is also verified.

If a node of the tree satisfies the candidate conditions 1, 2, 3 (of Definition 2) and the topology constraint (of Definition 3), then the node satisfies the isomorphism conditions. The candidate condition 4 of Definition 2 is a filtering test that often obviates the need for the substantial work to verify the topology constraint.

Since all nodes in the tree satisfy, for construction, the candidate conditions of Definition 2, the second step of the proposed implementation aims at searching any leaf of the tree satisfying the topology constraint. Verifying the topology constraint of all nodes of a path, for all paths of the tree is the bottleneck of the SubGI algorithm. *GRASS* implements such a verification in parallel in two phases: *flooding* and *hypersearch*.

A. The flooding phase

Starting from the root, *GRASS* calculates the maximum depth of the tree (called *flooding level*) whereby a complete and parallel BFS visit can be accomplished in one GPU kernel invocation and by assigning one path (from the root to a node at the flooding level) to a GPU thread. Each thread checks the topology constraint of the nodes along the assigned path and prunes the unsatisfying nodes (and the corresponding underling branches). The parallel visit and constraint check run in backtracking, i.e., starting from the nodes at the flooding level up to the root. This allows the set of paths to be automatically partitioned, by assigning one node at the flooding level to a thread.

The flooding level is calculated by considering the architectural limitations of the GPU device (i.e., maximum number of parallel running threads) and the maximum number of candidates at each tree level. Since *GRASS* implements a static order of query vertices, each level in the tree is associated to a query vertex (see Section III). This allows the maximum number of candidates at each tree level (`Candidate_number[level]`) to be calculated in the pre-processing phase. Given the maximum number of parallel threads of the GPU architecture (`Max_threads`), the flooding level is calculated as follows:

```
1: Flooding_level = Starting_level //= Level 1;
2: Candidate_counter = 1;
3: while Candidate_counter ≤ Max_threads do
4:     Candidate_counter *= Candidate_number[Flooding_level];
5:     Flooding_level++;
6: end while
```

The candidate counter increases monotonically during the tree construction. Figure 3 shows an example, in which $Max\_threads = 10$ (the actual value is generally around tens of thousands). The candidate number information is reported in the rightmost side. The resulting flooding level is level three.

Given the flooding level, x, the threads check the topology constraint of the nodes along the own path in parallel and with no need of any synchronization. Then, they store the results ($true$ if all nodes satisfy the constraint, $false$ otherwise) in a $bitset$ array. Each resulting $true$ corresponds to a *partial match*, that is, an isomorphism occurrence of the first x nodes of the query sequence μ in graph G. A thread ends the assigned job as soon as it finds an unsatisfying node on the path or if it concludes the path visit.

The flooding phase allows the visit to rapidly step down into the tree and to find enough partial matches as starting

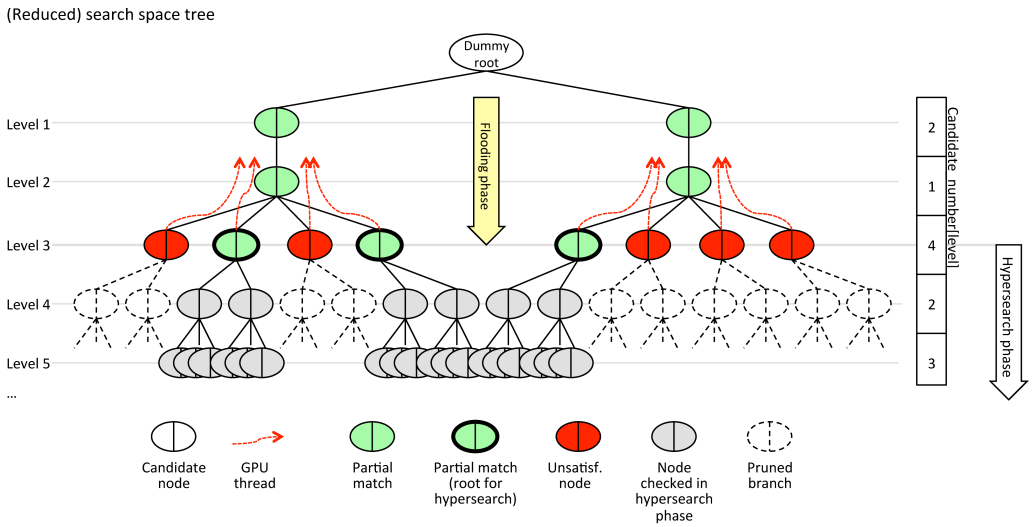

Fig. 3. Example of flooding phase on a (reduced) tree, given Maxthread = 10

points for the hypersearch phase. A large number of partial matches is relevant for having workload balancing in the most expensive phase of the whole visit (i.e., the hypersearch phase), as explained in the following section. The flooding phase provides the best results on high degree graphs, where the difference of the parallelism degree between two levels is meaningful. In addition, the higher the degree of G, the closer the flooding level to the root, the shorter the checked paths, and the more reduced the effect of workload imbalance in the early stage of the visit.

B. The hypersearch phase

The hypersearch phase consists of a parallel visit of the sub-trees whose roots are the nodes classified as partial matches in the flooding phase. Each of these nodes is assigned to a GPU thread block, which takes charge of the complete visit and topology check of the underlying sub-tree.

Figure 4 shows the main idea. $Block\_1$, $Block\_2$, and $Block\_3$ run in parallel the DFS visits of the sub-trees having the assigned partial match nodes as root. The DFS visit consists of iterative BFS visits, where each BFS visit is performed in parallel by the block threads.

A thread block performs a parallel BFS visit over the first levels of the sub-tree. Similarly to the flooding step, a manager thread calculates how many levels of the sub-tree can be completely covered by the block threads (i.e., the sub-tree level in which all nodes can be assigned to the own block thread). The target level is calculated by considering the block size (number of threads per block) and the maximum number of candidates per level. In the example, considering six threads per block, and the candidate number on the rightside, $Block\_1$ performs a BFS visit (in backtracking) from the nodes at level five to the assigned root at level three. $Block\_1$ carries on the visit by iterating a second BFS visit, by starting from the candidate nodes found at the previous BFS step. $Block\_2$ ends after one BFS visit, since all the visited paths result in being unsatisfying solutions. $Block\_3$ performs three sequential BFS visits. After the first BFS between level three and level five, it performs the second BFS visit between level five and level seven. Finally, it backtracks to level five to start the third and last BFS visit.

When a thread block concludes the sub-tree visit (either by providing isomorphism occurrences or by prematurely terminating for unsatisfying conditions of all nodes), the thread block takes charge of a new partial match.

The partial matches returned by the flooding phase are generally more than the number of thread blocks. This allows all the possible thread blocks to be run in parallel and, thus, to fully exploit the GPU thread parallelism. In addition, the dynamic assignment of partial match nodes to blocks and the independence among blocks allow the workload of the visit to be controlled.

V. Experimental Results

GRASS has been tested on three datasets of graphs, each one with different topological characteristics and size. Two datasets are biological graphs widely used as benchmarks [12], [33]. The first represents microbial protein interaction networks [41], where interactions among proteins have been inferred from functional genomic data evidences via the SRINI algorithm [42]. Microbial networks were uniformly labeled with distinct amount of labels, ranging from 1 to 64. The second is a set of protein contact maps, where edges among atoms represent physical and chemical bounds retrieved from the PDB database [43]. Microbial networks range from 1k to 4k vertices and are relatively large sparse graphs, with an average degree of 24. In contrast, contact maps have up to 800 vertices but are more dense, with and average degree of 50. In addition, a set of synthetic graphs were generated. A fixed number of vertices (from 100k to 10 millions) and the average degree (from 4 to 50) has been provided as input to the model, then edges have been assigned to vertices with a uniform distributed probability. The generated topologies were randomly labeled with several amounts of distinct labels. For real datasets, we used the original query set provided in [33], and we randomly extracted substructures from the synthetic

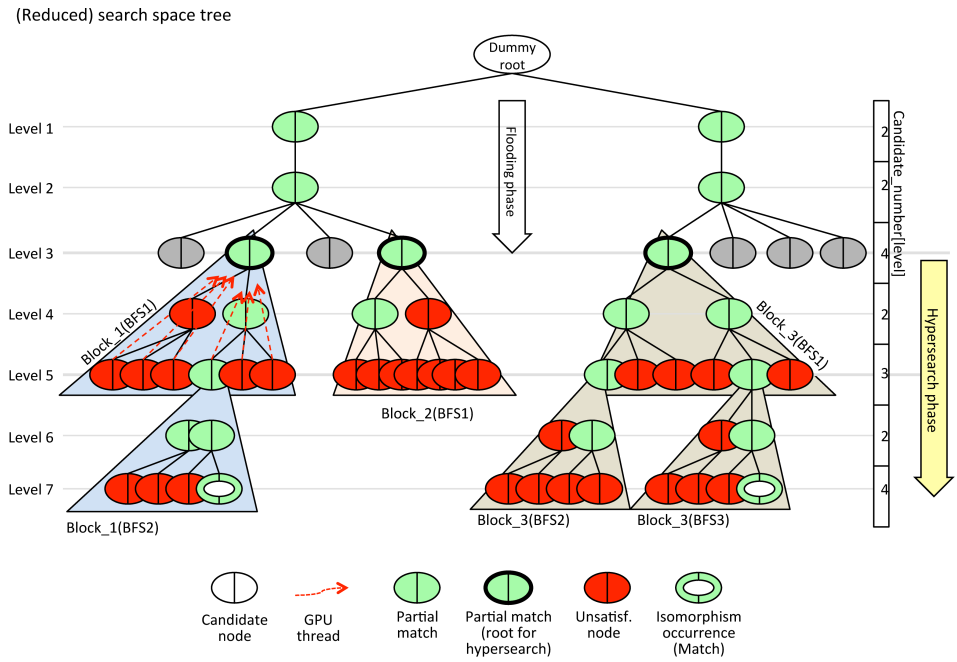

Fig. 4. Example of hypersearch with thread block size = 6.

topologies after the labeling phase. THe tested SubGI instances are composed on a query and the target graph from which the query has been extracted.

None of the GPU implementations for SubGI presented as related work is available (no public repository exists, no code released by the authors, or code not maintained anymore). The STWig solution presented in [36] is 2 orders of magnitude slower than the best (and more recent) sequential approaches (TurboISO and RI), as shown in [44] and [33]. Thus, we evaluated the performance of *GRASS* w.r.t. the sequential algorithms TurboISO and RI. The tests were run on a Intel Core i7-5960X 64bit hardware with 16 3GHz CPUs and 64Gb of host RAM and running a Ubuntu 16.04 LTS operating system, equipped with an NVIDIA GeForce GTX 980 Ti GPU card with 6Gb of RAM and running with the CUDA 8.0 toolkit. All the compared algorithms are implemented in C++.

Figure 5 shows the number of instances of the PPI dataset finished within a 10 minutes timeout of the three implementations. As expected, the lower the number of labels, the higher the computational requirement of the SubGI. This is due to the fact that the query nodes matches to an higher number of target elements. The number of finished instances generally tends to decrease together with the number of target labels. The *GRASS* performance are not affected by such a discriminant, except for unlabeled graphs (number of labels equal to 1), and it outperforms the other two tools with every label number. Another discriminant parameter for SubGI expensiveness is the query size, here reported in terms of number of vertices. Contrary to what can be expected, larger queries are not more difficult to be solved, since the number of matches usually decreases (not linearly) with the size of the structure [33]. 4-vertices queries produce a huge amount of matches that infer the algorithm performance. By combining the two charts, *GRASS* is able to solve any SubGI instance within the 10-minutes timeout except for unlabeled queries having 4 nodes, as opposite to the two serial solvers that reached the timeout in several circumstances. Regarding the contact map benchmarks, the number of finished instances for *GRASS* and RI was stably closed to 100%, while it was about 80% for TurboIso. Similar observations made for the PPI dataset can be done also for this case.

The speed-ups were calculated only for SubGI instances with a running time grater than 1 second for the competitor (RI or TurboIso). Figure 6 shows the speed-up obtained by running the three approaches over the real dataset and by grouping them by the number of distinct target labels. No RI run exceeded 1 second on PPI targets with 64 labels, thus the corresponding speed-up is not shown. Clearly 32 and 64 distinct labels reduce the complexity of SUbGI instances. *GRASS* provides the best speed-ups on unlabeled targets (1 label) and graphs with 24 labels. Speed-ups grouped by number of query vertices are shown in Figure 7. The minimum speed-ups were observed with 4-labels targets (2.4x vs RI) and 16-vertices queries (2.47x vs RI) on PPIs, 21-labels targets (2.8x vs TurboIso) and 64-vertices queries (3.09x vs RI) for contact maps. On the PPI dataset, the overall average speed-up of *GRASS* was 8.97x over RI and 32.1x over TurboIso. For the contact maps, the overall average speed-up of *GRASS* was 59x over RI and 61.2x over TurboIso. For what concerns the synthetic dataset, the overall average speed-up of *GRASS* was 2.5x over RI and 55.7x over TurboIso.

For what concerns the synthetic dataset (see Figure 8), the experiments were run with a 10 minutes timeout for 100k and 1m vertices target graphs, and a 30 minutes timeout was used for the target graphs with 10m vertices. This choice was due to the fact that *GRASS* increases the time for reading the target graph, and converting it into the data structures that are

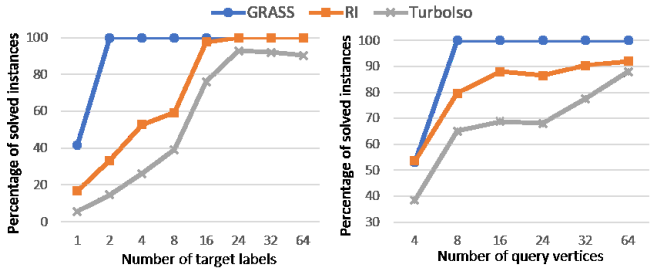

Fig. 5. Percentage of solved SubGI instances, within a 10 minutes timeout, by the three compared tools over the PPI dataset. Results are grouped by number of target labels and by number of query vertices, respectively on the left and right charts.

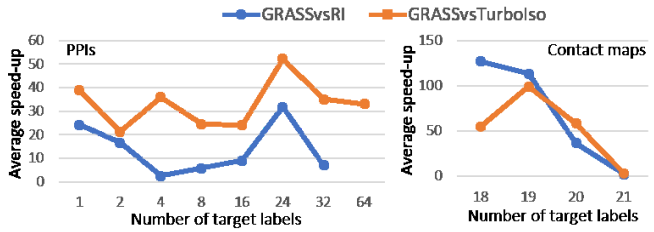

Fig. 6. Average speed-up of *GRASS* w.r.t. RI and TurboIso over the two real datasets (PPIs on the left side and contact maps on the right side). Averages are computed by grouping total running times of SubIG instances by number of target labels.

loaded into the GPU memory, from less than 1 second for 100k-vertices graph up to 10 minutes in 10 millions vertices targets. The limitation of *GRASS* for reading and converting graphs can be easily bypassed if input data is provided already in the form that *GRASS* uses for its computation. For this reason, the speed-up over the synthetic dataset were calculated by taking into account the effecting matching time, excluding reading and preparing phases. The matching time also includes loading and copy back of the data from the host machine to the GPU device.

The speed-up measurement has to be correlated with the number of finished instances, that *GRASS* maximizes in every situation w.r.t. the sequential algorithms. Such a calculation of the speed-up, by taking into account only the instances that the two compared solutions both finished within the timeout, is at the expense of the fastest algorithm, namely *GRASS*. A clear example is given by the decreasing speed-up of *GRASS* over TurboIso on the increasing size of the queries, in the synthetic dataset. In this case, the speed-up drops from 70x to 20x, but the percentage of finished instances of TurboIso drops from 95% to 23%.

VI. Conclusions

This paper presented *GRASS*, a parallel implementation of the SubGI algorithm for GPUs. *GRASS* implements a pre-processing phase, in which heuristics are applied to sensibly reduce the search space and, as a consequence, to support large graphs also when run on low-end GPU devices. It combines the DFS and BFS visiting strategiesc through two different kernels, to better exploit the massive thread parallelism and

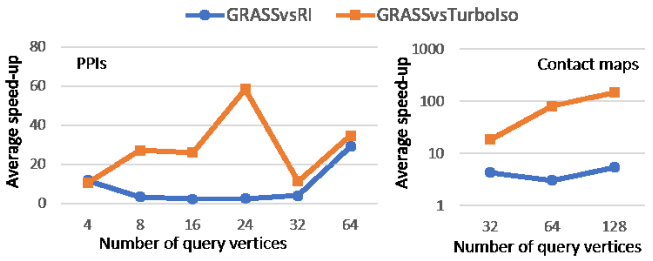

Fig. 7. Average speed-up of *GRASS* w.r.t. RI and TurboIso over the two real datasets (PPIs on the left side and contact maps on the right side). Averages are computed by grouping total running times of SubIG instances by number of query vertices.

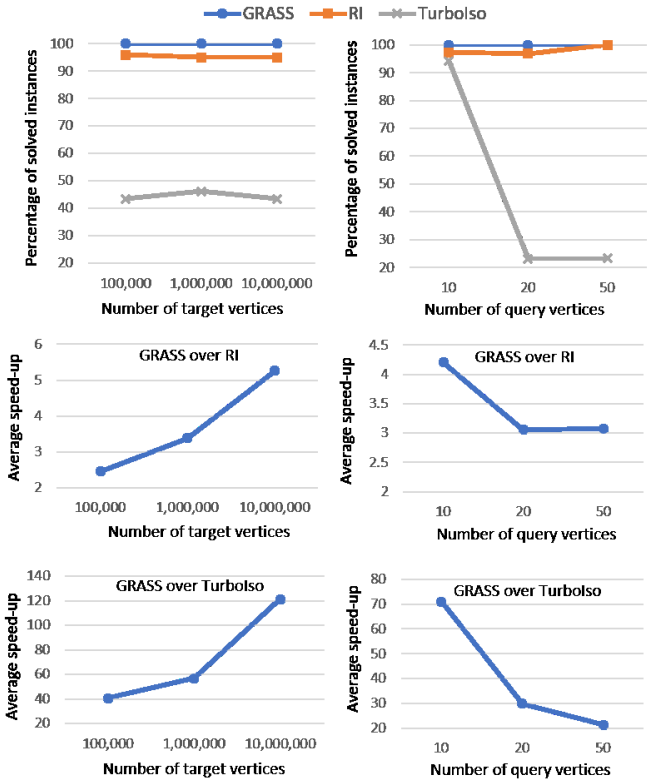

Fig. 8. Performances of *GRASS* RI and TurboIso over the synthetic dataset. On the left column, results are grouped by number of target vertices, and on the rights column results are grouped by number of query vertices. Charts on the top side show the number of finished SubGI instances by the three approaches. Middle-side and bottom-side charts show speed-up of *GRASS* over Ri and TurboIso, respectively.

to optimize the workload balancing during the visit of non-homogeneous topologies of actual graphs. The paper presented the experimental results obtained by comparing the proposed approach with sequential implementations on real and synthetic datasets of different sizes and characteristics. The results have been also analyzed to understand the efficiency and the limitations of the proposed approach and how they are related to the characteristics of the involved graphs.

Acknowledgment

This work has been partially supported by the following projects: GNCS-INDAM, Fondo Sociale Europeo, National Research Council Flagship Projects Interomics, JOINT

PROJECTS 2016-JPVR16FNCL, and JOINT PROJECTS 2017-B33C17000440003. This work has been partially supported by the project of the Italian Ministry of Education, Universities and Research (MIUR) "Dipartimenti di Eccellenza 2018-2022".

REFERENCES

[1] B. Schwikowski, P. Uetz, and S. Fields, "A Network of Protein-Protein Interactions in Yeast," *Nature Biotechnology*, vol. 18, no. 12, pp. 1257–1261, 2000.

[2] R. D. Finn, B. L. Miller, J. Clements, and A. Bateman, "IPFAM: a Database of Protein Family and Domain Interactions Found in the Protein Data Bank," *Nucleic Acids Research*, vol. 42, no. D1, pp. D364–D373, 2014.

[3] G. W. Bemis and M. A. Murcko, "The Properties of Known Drugs. 1. Molecular Frameworks," *Journal of Medicinal Chemistry*, vol. 39, no. 15, pp. 2887–2893, 1996.

[4] I. Gutman, L. Popović *et al.*, "Graph Representation of Organic Molecules Cayley's Plerograms vs. His Kenograms," *Journal of the Chemical Society, Faraday Transactions*, vol. 94, no. 7, pp. 857–860, 1998.

[5] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The Anatomy of the Facebook Social Graph," *arXiv preprint arXiv:1111.4503*, 2011.

[6] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network Motifs: Simple Building Blocks of Complex Networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[7] T. Kindt, S. Morse, E. Gotschlich, and K. Lyons, "Structure-based Strategies for Drug Design and Discovery," *Nature*, vol. 352, p. 581, 1991.

[8] K. Juszczyszyn, P. Kazienko, and K. Musiał, "Local Topology of Social Network Based on Motif Analysis," in *Knowledge-Based Intelligent Information and Engineering Systems*. Springer, 2008, pp. 97–105.

[9] S. Mangan and U. Alon, "Structure and Function of the Feed-forward Loop Network Motif," *Proceedings of the National Academy of Sciences*, vol. 100, no. 21, pp. 11 980–11 985, 2003.

[10] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 788–799, May 2012.

[11] W.-S. Han, J. Lee, and J.-H. Lee, "Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 337–348.

[12] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A Subgraph Isomorphism Algorithm and its Application to Biochemical Data," *BMC Bioinformatics*, vol. 14, no. 7, pp. 1–13, 2013.

[13] C. Solnon, "All Different-based Filtering for Subgraph Isomorphism," *Artificial Intelligence*, vol. 174, no. 12-13, pp. 850–864, 2010.

[14] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 804–818, 2018.

[15] J. R. Ullmann, "Bit-vector Algorithms for Binary Constraint Satisfaction and Subgraph Isomorphism," *Journal of Experimental Algorithmics*, vol. 15, pp. 1.1–1.64, February 2011.

[16] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[17] V. Carletti, P. Foggia, A. Greco, A. Saggese, and M. Vento, "Comparing performance of graph matching algorithms on huge graphs," *Pattern Recognition Letters*, 2018.

[18] A. Aparo, V. Bonnici, G. Micale, A. Ferro, D. Shasha, A. Pulvirenti, and R. Giugno, "Simple pattern-only heuristics lead to fast subgraph matching strategies on very large networks," in *International Conference on Practical Applications of Computational Biology & Bioinformatics*. Springer, 2018, pp. 131–138.

[19] C. McCreesh, P. Prosser, C. Solnon, and J. Trimble, "When subgraph isomorphism is really hard, and why this matters for graph databases," *Journal of Artificial Intelligence Research*, vol. 61, pp. 723–759, 2018.

[20] X. Ren, J. Wang, N. Franciscus, and B. Stantic, "Experimental clarification of some issues in subgraph isomorphism algorithms," in *Asian Conference on Intelligent Information and Database Systems*. Springer, 2018, pp. 71–80.

[21] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 117–128.

[22] F. Busato and N. Bombieri, "BFS-4K: an efficient implementation of BFS for kepler GPU architectures," *IEEE Transactions on Parallel Distributed Systems*, vol. 26, no. 7, pp. 1826–1838, 2015.

[23] P. Harish and P. J. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC'07, 2007, pp. 197–208.

[24] G. J. Katz and J. T. Kider, Jr, "All-pairs Shortest-paths for Large Graphs on the GPU," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2008, pp. 47–55.

[25] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul. 2008.

[26] B. R. Gaster and L. Howes, "Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck?" *IEEE Computer*, vol. 45, no. 8, pp. 42–52, 2012.

[27] F. Busato and N. Bombieri, "Efficient load balancing techniques for graph traversal applications on gpus," in *European Conference on Parallel Processing*. Springer, 2018, pp. 628–641.

[28] F. Busato, O. Green, N. Bombieri, and D. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus," *IEEE Proc. High Performance Extreme Computing (HPEC), Waltham, MA*, 2018.

[29] F. Wang, J. Dong, and B. Yuan, "Graph-Based Substructure Pattern Mining Using CUDA Dynamic Parallelism," *Intelligent Data Engineering and Automated Learning ? IDEAL 2013 Lecture Notes in Computer Science*, vol. 8206, pp. 342–349, 2013.

[30] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *International Conference on Database Systems for Advanced Applications*. Springer, 2015, pp. 299–315.

[31] J. Ullmann, "An Algorithm for Subgraph Isomorphism," *Journal of the Association for Computing Machinery*, vol. 23, pp. 31–42, 1976.

[32] C. Lecoutre, *Constraint Networks: Techniques and Algorithms*. ISTE/Wiley, 2009.

[33] V. Bonnici and R. Giugno, "On the variable ordering in subgraph isomorphism algorithms," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 14, no. 1, pp. 193–203, 2017.

[34] R. Kessl, N. Talukder, P. Anchuri, and M. J. Zaki, "Parallel graph mining with gpus," in *Proceedings of the 3rd International Conference on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications-Volume 36*. JMLR. org, 2014, pp. 1–16.

[35] X. Yan and J. Han, "GSPAN: Graph-Based Substructure Pattern Mining," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, ser. ICDM '02, 2002, pp. 721–724.

[36] X. Lin, R. Zhang, Z. Wen, H. Wang, and J. Qi, "Efficient subgraph matching using gpus," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8506 LNCS, pp. 74–85, 2014.

[37] J. J. McGregor, "Relational consistency algorithms and their application in finding subgraph and graph isomorphisms," *Information Sciences*, vol. 19, no. 3, pp. 229–250, 1979.

[38] B. Yang, K. Lu, Y.-h. Gao, X.-p. Wang, and K. Xu, "Gpu acceleration of subgraph isomorphism search in large scale graph," *Journal of Central South University*, vol. 22, no. 6, pp. 2238–2249, 2015.

[39] M.-Y. Son, Y.-H. Kim, and B.-W. Oh, "An efficient parallel algorithm for graph isomorphism on gpu using cuda," *Int. J. of Engineering and Technology (IJET)*, vol. 7, no. 5, pp. 1840–1848, 2015.

[40] F. Bacchus and P. van Run, "Dynamic Variable Reordering in CSPs," in *CP '95 Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, 1995, pp. 258–275.

[41] "http://www.ncbi.nlm.nih.gov/pubmed/16899655," 2006.

[42] B. S. Srinivasan, A. F. Novak, J. A. Flannick, S. Batzoglou, and H. H. McAdams, "Integrated protein interaction networks for 11 microbes," in *Research in Computational Molecular Biology*. Springer, 2006, pp. 1–14.

[43] "Protein Data Bank. http://www.rcsb.org/pdb/."

[44] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," in *Proceedings of the 39th international conference on Very Large Data Bases*, ser. PVLDB'13, 2013, pp. 133–144.